Lecture 6

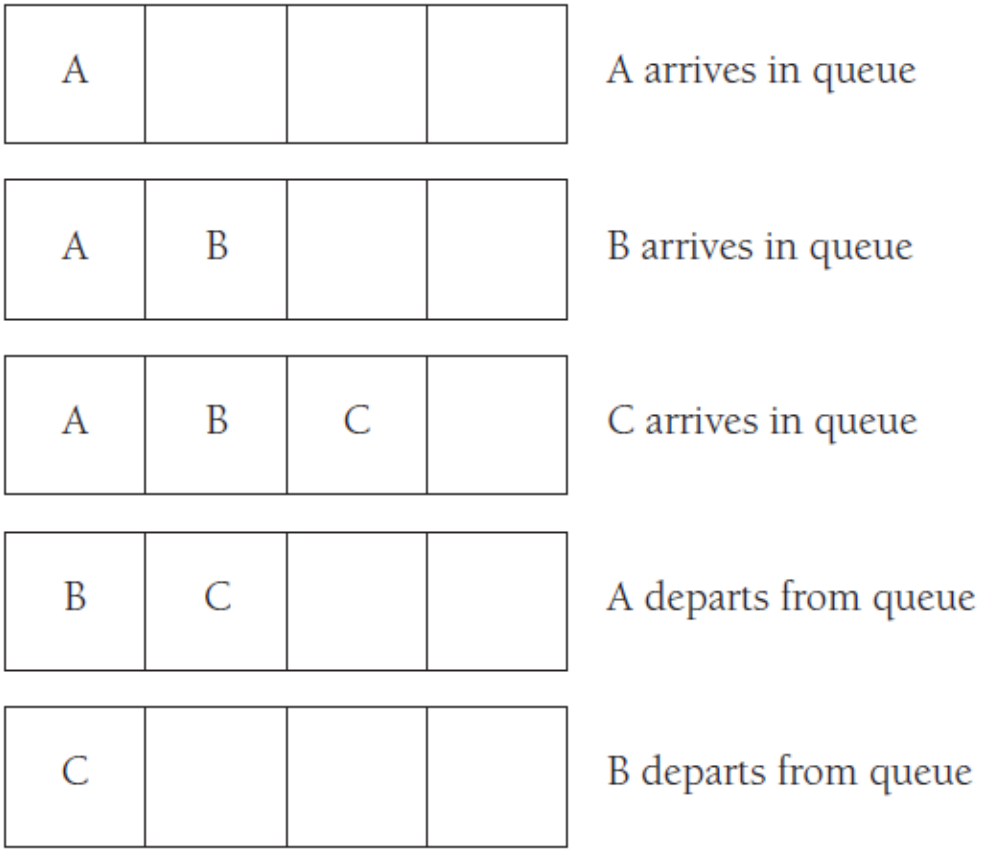
Queues

Queue data structure characteristics

- A queue is a data structure where data enters at the rear of a list and is removed from the front of the list.
- It's a First In First Out (FIFO) data structure
- Some queues applications:
  - Queues are used to order processes submitted to an operating system or a print spooler
  - Simulation applications use queues to model customers waiting in a line.

Queue Operations

- Adding an element to the queue is called “Enqueue”
- Removing an item from the queue is called “Dequeue”
- Viewing, but not removing the first item from the queue is called “Peek”
- Other properties like “Count” , Clear, and “IsEmpty” are also common properties in queue implementation



Queue implementation

For maintaining simplicity, no check for empty queue was made for the DeQueue and Peek functions

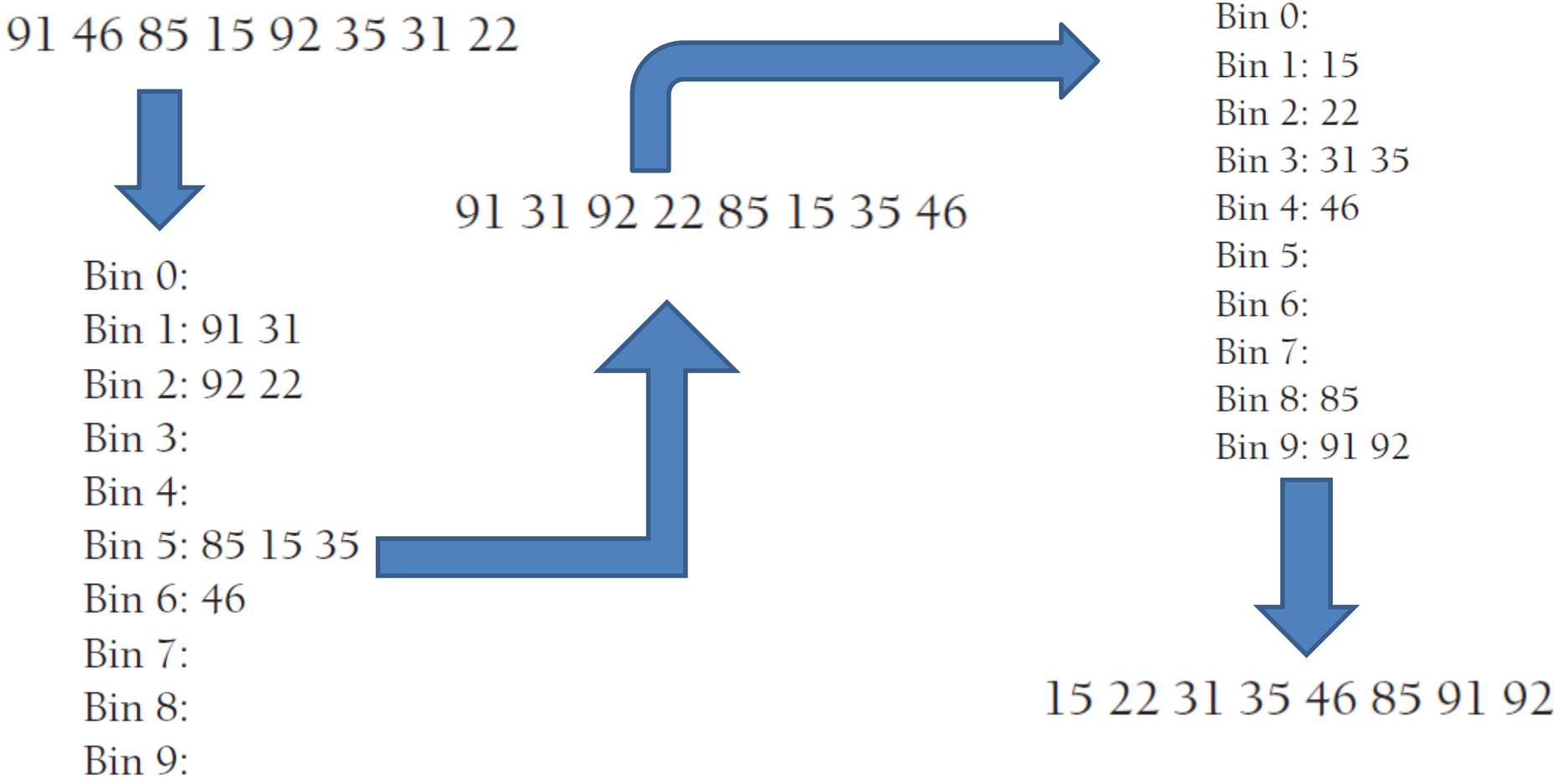
```
class OurQueue<T>
{
    private List<T> list;
    public OurQueue()
    {
        list = new List<T>();
    }

    public int Count
    {
        get
        {
            return list.Count;
        }
    }
    public void EnQueue(T item)
    {
        list.Add(item);
    }
    public T DeQueue()
    {
        // no safty code for empty queue for simplicty
        T element = list[0];
        list.RemoveAt(0);
        return element;
    }
    public void clear()
    {
        list.Clear();
    }
    public T Peek()
    {
        // no safty code for empty queue for simplicty
        return list[0];
    }
}
```

.NET Framework queue

```
// A generic queue (System.Collection.Generic)
//Creating a queue using the default settings
Queue<int> numbers = new Queue<int>();
// Creating a generic queue with initial capacity and growth rate
Queue<Customer> customers= new Queue<Customer>(32,3);
Customer cust1=customers.Dequeue();
// A non generic queue (System.Collections)
// Creating a queue with the default settings
Queue myQueue=new Queue();
// Creating a queue and specifying its initial capacity
Queue myQueue = new Queue(100);
// Creating a queue and specifying its capacity and its growth factor
Queue myQueue = new Queue(32, 3);
myQueue.Enqueue(5);
Int x=(int)myQueue.DeQueue();
```

Queue Application: Radix sort



Radix sort implementation: soring function

```
private static Queue<int> RadixSort(Queue<int> numbers, int sortingDigitWeight)
{
    Queue<int>[] sortingQueues = new Queue<int>[10];
    // creating sorting queues
    for (int i = 0; i < 10; i++) sortingQueues[i] = new Queue<int>();
    // sort into bins
    // assuming only two digits integer number
    int binIndex;
    while (numbers.Count > 0)
    {
        if (sortingDigitWeight == 1)
            binIndex = numbers.Peek() % 10;
        else
            binIndex = numbers.Peek() / 10;
        sortingQueues[binIndex].Enqueue(numbers.Dequeue());
    }
    //collecting
    Queue<int> collectingQueue = new Queue<int>();
    for (int i = 0; i < 10; i++)
        while (sortingQueues[i].Count > 0) collectingQueue.Enqueue(sortingQueues[i].Dequeue());
    return collectingQueue;
}
```

Radix sort implementation: Main function

```
static void Main()
{
    // Using our queue implementation to sort numbers using radix sort
    // entring 10 numbers each is less that 100 (maximum 2 digits)to sort
    Queue<int> numbersQueue = new Queue<int>();
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("Enter number {0}(int 0-99):", i);
        numbersQueue.Enqueue(int.Parse(Console.ReadLine()));
    }
    //sort on digit one
    numbersQueue = RadixSort(numbersQueue, 1);
    //sort on digit two
    numbersQueue = RadixSort(numbersQueue, 10);
    // print sorted numbers
    Console.WriteLine("Sorted numbers:");
    while (numbersQueue.Count > 0) Console.WriteLine(numbersQueue.Dequeue());
    Console.WriteLine("Press any key to continue:");
    Console.Read();
}
```

Report Discussion

Last lecture report:

- DOTNET Stack class methods and properties
- The stack role in function calls

New report:

- DOTNET Queue class methods and properties
- The queue role in Operating systems

Priority queue

Element has the heighest priority (smallest priority number) is removed when DequeuePriority is executed

In the same time, for elements with the same priority: FIFO is retained

```
class PriorityNameQueue : Queue<PriorityName>
{
    public PriorityName DeQueuePriority()
    {
        // assuming the smallest priority is the heighest
        PriorityName[] items;
        int x, minindex;
        PriorityName min;
        items = this.ToArray();
        min = items[0];
        minindex = 0;
        for (x = 1; x < items.Length; x++)
            if (items[x].Priority < min.Priority)
            {
                min = items[x];
                minindex = x;
            }
        this.Clear();
        for (x = 0; x < items.Length; x++)
            if (x != minindex)
                this.Enqueue(items[x]);
        return items[minindex];
    }
}
class PriorityName
{
    public PriorityName(string name, int priority)
    { Name = name; Priority = priority; }
    public string Name;
    public int Priority = 0;
}
```

Priority queue application

```
class Example2
{
    static void Main()
    {
        PriorityName pn;
        PriorityNameQueue pnq = new PriorityNameQueue();
        for (int i = 0; i < 3; i++)
        {
            pn = new PriorityName("", 0);
            Console.WriteLine(" Enter name {0} :", i);
            pn.Name = Console.ReadLine();
            Console.WriteLine(" Enter priority {0} :", i);
            pn.Priority = int.Parse(Console.ReadLine());
            pnq.Enqueue(pn);
        }
        Console.WriteLine(" Extrracting names from the queue in priority:");
        while (pnq.Count > 0)
        {
            PriorityName pnExtracted = pnq.DeQueuePriority();
            Console.WriteLine("Name: {0}  Priority: {1} ", pnExtracted.Name, pnExtracted.Priority);
        }
        Console.WriteLine("Press any key to continue:");
        Console.Read();
    }
}
```

Exercises

Now you can solve sheet 6